DOCUMENTO GUÍA PARA LA ENSEÑANZA DE LA PROGRAMACIÓN FUNCIONAL EN EL PROGRAMA ISC-UTP

EDISON ANDRES MONTOYA LONDOÑO

DANIEL FERNANDO LARGO GUERRERO

UNIVERSIDAD TECNOLÓGICA DE PEREIRA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

PEREIRA

2018

DOCUMENTO GUÍA PARA LA ENSEÑANZA DE LA PROGRAMACIÓN FUNCIONAL EN EL PROGRAMA ISC-UTP

EDISON ANDRES MONTOYA LONDOÑO

DANIEL FERNANDO LARGO GUERRERO

DIRECTOR

Msc. CARLOS AUGUSTO MENESES

UNIVERSIDAD TECNOLÓGICA DE PEREIRA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

PEREIRA

2018

Resumen

En este documento se presenta una monografía que reúne el estado del arte de la programación funcional para que sirva como documento de consulta para los estudiantes del curso de programación I del programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira. En él se relaciona el paradigma de programación que se fundamenta en el uso de funciones para la construcción de programas de computadora y que representa una de las alternativas a partir de las cuales, se puede orientar el pensamiento para encontrar soluciones a problemas. También se hace referencia a algunos de los lenguajes que han sido parte de la evolución de la programación funcional y a la fundamentación lógica-deductiva que da origen a la evaluación de las funciones de tal manera que puedan ser consideradas como factibles para ejecutarse apropiadamente en un sistema computacional.

DOCUMENTO GUÍA PARA LA ENSEÑANZA DE LA PROGRAMACIÓN FUNCIONAL EN EL PROGRAMA ISC-UTP

TABLA DE CONTENIDO

1. INTRODUCCIÓN

Mediante la programación de computadoras se puede crear una serie de instrucciones para desarrollar una tarea o darle solución a un problema con mayor agilidad y precisión. Estas ventajas permiten que su uso sea una buena opción para mejorar y automatizar muchos de los procesos que se llevan a cabo y en mayor medida, aquellos que se componen de tareas repetitivas.

La programación se puede abordar desde múltiples paradigmas, dentro de los cuales se encuentran el declarativo, imperativo, funcional, entre otros. Cada uno de ellos aborda problemas de forma diferente y con algunas características especiales, facilitando su interpretación desde ángulos específicos.

En este documento se enfatizará en la programación funcional, que permite ligar las matemáticas con la programación, de tal manera que se asocie como un recurso que permite la solución de cierto tipo de problemas.

2. GENERALIDADES

2.1. DEFINICION DEL PROBLEMA

La programación funcional forma parte fundamental de la historia del pensamiento orientado a la solución de problemas [1]. Representa una base en la que los estudiantes se puedan apoyar para adquirir experiencia en la elaboración de algoritmos y en la traducción de instrucciones que sean fácilmente interpretadas por máquinas mediante el uso de lenguajes programación.

La forma en que esta programación es orientada influye, tanto en los tiempos de asimilación de conceptos [2], como en la evolución del aprendizaje de los estudiantes con respecto a este tema y la promoción del uso de este tipo de paradigmas para abordar situaciones. De ahí que se plantee la siguiente pregunta y que se trabaje en torno a darle una respuesta.

¿Es posible construir un documento guía como base para la enseñanza de la programación funcional en ISC-UTP?

2.2. OBJETIVO GENERAL

Elaborar una monografía que reúna el estado del arte de la programación funcional para que sirva como documento de consulta para los estudiantes de programación I de la Universidad Tecnológica de Pereira.

2.3. OBJETIVOS ESPECÍFICOS

- Recolectar documentos bibliográficos para la elaboración de la monografía.
- Plantear el contenido temático del documento final.
- Desarrollar los contenidos del documento final, orientando hacia la especificación de los conceptos encontrados.

2.4. JUSTIFICACIÓN

Nos proponemos entonces indagar sobre la programación funcional, debido a que es uno de los primeros acercamientos que se tienen con respecto a los grandes paradigmas de programación existentes y que articula una gran base matemática para la resolución de problemas. Además, representa un gran desafío en la forma de desarrollo del pensamiento, ya que se deben modelar soluciones a partir de la asimilación y uso de conceptos matemáticos y lógicos apropiados para cada tipo de problema, lo que normalmente causa dificultad en los estudiantes, más aún, cuando no han sido formados bajo una guía clara y consecuente.

El documento es una herramienta con conocimiento base en donde los estudiantes encontrarán los fundamentos de la programación funcional, lo cual ayudará como su primer acercamiento con este tema y representará una alternativa para aclarar dudas por medio de ejemplos que permitan al estudiante apropiarse de los conceptos.

También servirá en el proceso de enseñanza de los docentes ya que será un material de apoyo a temas principales desarrollados como parte del curso de programación funcional, además de traer consigo la identificación de un lenguaje de programación que puede ser usado pare ello.

2.5. METODOLOGÍA APLICADA

- La metodología que se usó para la construcción del documento incluye:
- Consulta en libros, documentos y revistas, las investigaciones que se han desarrollado en torno a la programación funcional.
- Análisis de la información recolectada con el fin de establecer conceptos que ayuden a identificar el funcionamiento de la programación funcional y sus características.
- Análisis de los referentes matemáticos que dieron lugar a dicha programación.
- Consultas en diferentes medios disponibles para ello, como libros, internet, entre otros, los lenguajes de programación que estén enfocados o hayan sido desarrollados bajo el esquema de programación funcional.
- Presentar en forma de conclusiones los aportes encontrados a través del estudio realizado.

3. ESTADO DEL ARTE

3.1. MARCO HISTÓRICO

La programación funcional desde tiempos pasados siempre ha existido pero no de una manera tan clara como ahora, al retroceder en el tiempo, se puede encontrar que se dieron los primeros indicios cuando el filósofo Platón salió un buen día con su pupilo y durante el paseo Platón le pregunta a su pupilo: ”Cuál de esas dos personas paradas le parece más alta? ”, entonces el hombre miró hacia donde se encontraban estas dos personas y dijo: “Son más o menos de la misma altura” a lo que Platón responde: “Qué quieres decir con más o menos? ”. El joven contestó: “Desde esta distancia se ven igual, tal vez si estuviera más cerca vería alguna diferencia”. Platón dijo: “Entonces, si ninguna cosa es perfectamente igual a otra en este mundo, ¿cómo crees que entendemos el concepto de equidad ‘perfecta’?” El joven se quedó perplejo.

Es así como nació el primer esfuerzo por entender la naturaleza de las matemáticas. Platón sugirió que todo en nuestro mundo es solo una aproximación de la perfección. Además, se dio cuenta de que entendemos el concepto de perfección, aunque no la hayamos visto.

Después, Gottfried Leibniz creó la máquina mecánica de cálculo en el siglo XVII. Esta máquina fue un primer prototipo del dispositivo soñado por Leibniz: una máquina capaz de manipular símbolos y determinar si una frase matemática era o no un teorema, es decir, si una proposición que partía de un supuesto (hipótesis), afirma una verdad que no es evidente por sí misma.

En el año 1900, David Hilbert, un matemático alemán, propuso en un congreso una selección de 10 problemas, tomados de una lista de 23, que él consideraba los problemas matemáticos más relevantes a resolver en el siglo. Esto le permitió a la computación formar parte de los mecanismos empleados para probar la solución a algunos de estos problemas.

Fue gracias a la curiosidad de algunas mentes importantes como Alan Turing, John Von Neumann, Kurt Gödel y Alonzo Church, que se lograron grandes avances en la computación, ya que su curiosidad e interés en los sistemas formales y por responder preguntas sobre computación permitieron el desarrollo de grandes investigaciones. Algunas de las preguntas que motivaron grandes avances son las siguientes:

"Si tuviéramos máquinas con infinito poder computacional, ¿qué problemas podríamos resolver?, ¿podrían darse soluciones automáticamente?, ¿quedarían algunos problemas sin resolver?, ¿cuáles y por qué?, ¿podrían máquinas con diferentes diseños ser iguales en potencia?".

De las investigaciones mencionadas anteriormente resultó el desarrollo del sistema formal llamado ***"Cálculo lambda (λ)=función"*** que tuvo lugar en la década de 1930, y que fue llevado a cabo por el matemático llamado Alonzo Church [3]. Este sistema tenía su base en el estudio y uso de funciones, sus aplicaciones y la recursión y es considerado base fundamental en el paradigma de programación funcional.

Alan Turing también realizó un trabajo similar. Desarrolló un formalismo diferente (ahora conocido como la máquina de Turing), y lo usó para llegar a conclusiones similares a las de Alonso. Más tarde se demostró que la máquina de Turing y el cálculo lambda son equivalentes en potencia.

A finales de la década de 1950, el profesor del MIT, John McCarthy se interesó en el trabajo de Alonzo Church. En 1958 presentó el lenguaje de procesamiento de listas (Lisp). Una implementación del cálculo lambda que trabajaba en computadoras con arquitectura Von Neumann. En 1973, un grupo de programadores del Laboratorio de Inteligencia Artificial del MIT desarrollaron hardware al que llamaron "Máquina Lisp", una implementación nativa en hardware del cálculo lambda de Alonzo, que ya no usaba la arquitectura de Von Neumann.

Reseña de los lenguajes funcionales

- **Lips.**

La notación original de McCarthy usaba "expresiones M" en corchetes, las cuales serían traducidas a expresiones S. Como un ejemplo, la expresión M **car[cons[A,B]]** es equivalente a la expresión **S (car (cons A B))**. Una vez que Lisp fue implementado, los programadores rápidamente eligieron usar expresiones S, y las expresiones M fueron abandonadas. Lisp fue implementado primero por Steve Russell en un computador IBM 704. Russell había leído el artículo de McCarthy, y noto que la función eval de Lisp podía ser implementada en código de máquina. El resultado fue un intérprete de Lisp funcional que podía ser usado para correr programas Lisp, o más correctamente, "evaluar expresiones Lisp". El primer compilador completo de Lisp, fue implementado en 1962 por Tim Hart y Mike Levin en el MIT. Este compilador introdujo el modelo Lisp de compilación incremental, en el cual las funciones compiladas e interpretadas se pueden entremezclar libremente. El lenguaje en los memos de Hart y Levin es mucho más cercano al estilo moderno de Lisp que el anterior código de McCarthy.

- **Lisp y su relación con la inteligencia artificial.**

Fue usado como la implementación del lenguaje de programación Micro Planner que fue la fundación para el famoso sistema de AI SHRDLU [4]. En los años 1970, a medida que la investigación del AI engendró descendientes comerciales, el desempeño de los sistemas Lisp existentes se convirtió en un problema creciente.

Lisp era un sistema difícil de implementar con las técnicas de compilador y hardware común de los años 1970. Las rutinas de recolección de basura, desarrolladas por el entonces estudiante graduado del MIT, Daniel Edwards, hicieron práctico correr Lisp en sistemas de computación de propósito general, pero la eficacia todavía seguía siendo un problema.

Peter J. Landin describió ISWIM que es una notación algorítmica, si bien nunca fue implementado, su influencia fue decisiva en el desarrollo de la programación funcional y se pueden contar los lenguajes **SASL**, **Miranda** y **ML** como sus sucesores.

- **ML**

ML como sucesor de ISWIM fue creado por Robin Milner y otros, a finales de los años 1970 en la Universidad de Edimburgo. Se considera un lenguaje incompleto porque permite programar imperativamente con efectos colaterales.

Entre las características de ML se incluyen evaluación por valor, álgebra de funciones, manejo automatizado de memoria por medio de recolección de basura, polimorfismo parametrizado, análisis de estático de tipos, inferencia de tipos, tipos de datos algebraicos, llamada por patrones y manejo de excepciones.

Los lenguajes de la familia ML se aplican principalmente en el diseño y manipulación de lenguajes de programación (compiladores, analizadores, demostradores de teoremas), así como en bioinformática, sistemas financieros, protocolos de sincronización, etc.

En la actualidad varios lenguajes de la familia ML están disponibles, principalmente **Standard ML (SML)** y **Ocaml (Ocaml)**. Varios conceptos y la estructura de ML han influido en el diseño de otros lenguajes, tales como **Cyclone** y **Nemerle**.

- **Scheme**

En 1975, Gerald Jay Sussman y Guy Lewis Steel desarrollaron un intérprete motivados por entender y descifrar algunos conceptos y la aplicabilidad del trabajo desarrollado por Carl Hewit. Este interprete se conocería inicialmente como Schemer y posteriormente como Scheme [5].

Pero ¿Cuál es la relación del trabajo de Hewit y las funciones?

Pues bien, la teoría de Hewit se conoce como "La teoría de los actores como un modelo computacional", en esta teoría los objetos son entidades computacionalmente activas que pueden recibir y reaccionar a los mensajes. Estos objetos y los mensajes que intercambian fueron denominados actores.

En su intento por entender y experimentar con esta teoría, Sussman y Steel se dieron cuenta que el paso de los mensajes se podía expresar sintácticamente de la misma forma en que se invoca una función, con la diferencia que en la función se retornaba un valor y en el actor no retornaba nada, sino que más bien invocaba una continuación, es decir a otro actor. A partir de esto y usando MacLisp se pudo construir Scheme.

Posteriormente se empezó a desarrollar y publicar más investigaciones en donde se hacía alusión a los beneficios de Scheme. Dentro de estas investigaciones está el desarrollo de un nuevo compilador de Scheme llamado RABBIT que fue el resultado de la Maestría de Steel, y cuyo propósito era sustentar que Scheme podía soportar más paradigmas de programación adicionales al funcional, tales como Orientado a objetos e Imperativo.

Dentro de las ventajas iniciales del uso de Scheme se resalta que suministro una plataforma operacional que permitió a los teóricos realizar experimentos, de la misma manera que permitió hacer la semántica denotacional más accesible a los desarrolladores.

Desde 1981 y 1982 se ha empezado a usar Scheme como un medio para impartir curso de licenciaturas en diferentes universidades dentro de las que se encuentra la de Indiana, MIT y Yale.

- **Miranda**

Entre 1985 y 1986 fue desarrollado **Miranda,** un lenguaje de programación cuyo principal objetivo era implementar una versión comercial de un lenguaje de programación funcional. Fue el primer lenguaje puramente funcional en ser destinado al uso comercial y no al académico.

- **Erlang**

En los años 80 Erlang toma un lugar en los laboratorios de Ciencias de Computación de la Compañía de telefonía sueca Ericsson, surgió del intento de desarrollar un lenguaje de programación de alto nivel, y con capacidad para afrontar proyectos, especialmente de Telecomunicaciones. Para el diseño de Erlang se analizaron alrededor de 300 lenguajes de programación existentes, de estos lenguajes se seleccionaron: Lisp, Prolog, y otros, de los cuales se tomaron las características más relevantes [6].

- **Haskell**

En septiembre de 1987, se celebró la conferencia FPCA en la que se decidió formar un comité internacional que diseñase un nuevo lenguaje puramente funcional de propósito general denominado Haskell.

Con **Haskell** se pretendía unificar las características más importantes de los lenguajes funcionales como las funciones de orden superior, evaluación perezosa, inferencia estática de tipos, tipos de datos definidos por el usuario, encaje de patrones y listas por comprensión [7]. El lenguaje incorporaba, además, Entrada/Salida puramente funcional y definición de arrays por comprensión. Durante casi 10 años aparecieron varias versiones del lenguaje Haskell, hasta que en 1998 se decidió proporcionar una versión estable del lenguaje, que se denominó Haskell98.

- **Scala**

Martin Odersky diseño **Scala** en el 2003, Scala es un lenguaje multi-paradigma diseñado para expresar patrones de programación comunes de una forma concisa, elegante, y de tipado seguro. Integra fácilmente características de lenguajes orientados a objetos y funcionales [8].

Después de 4 años de haber aparecido Scala nace Clojure de dialecto de Lips y creado por Rich Hickey. Hace un énfasis en eliminar la complejidad de la programación concurrente en la programación funcional.

- **Rust**

Rust es un lenguaje de programación compilado, de propósito general y multiparadigma que está siendo desarrollado por Mozilla. Ha sido diseñado para ser "un lenguaje seguro, concurrente y práctico". Es un lenguaje de programación que soporta programación funcional pura, por procedimientos, imperativa y orientada a objetos.

Rust está diseñado para que se pueda desarrollar software para sistemas, donde la interacción con el usuario es poca o nula; Excelente para aplicaciones con el modelo cliente-servidor [9].

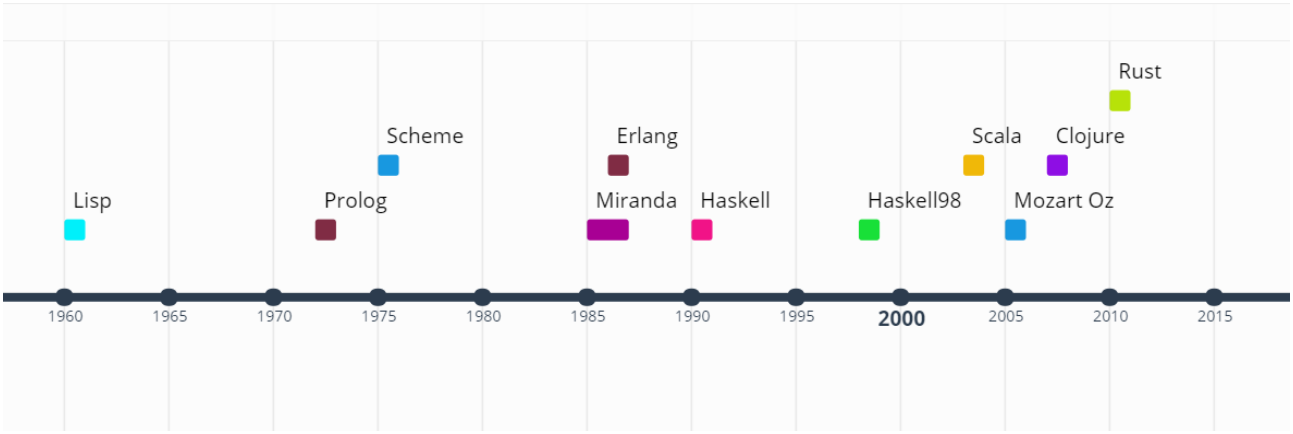
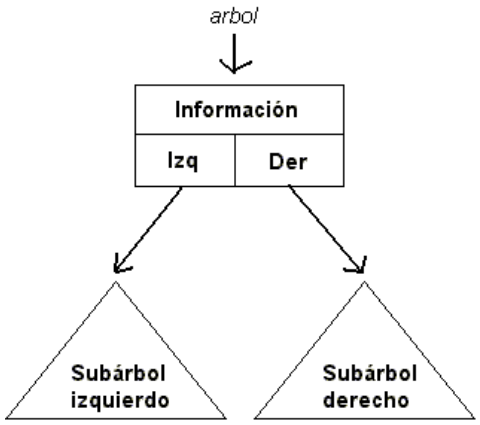


Ilustración 1. Linea del tiempo lenguajes funcionales

3.2. MARCO CONCEPTUAL

- **Algoritmo:** Un algoritmo es una secuencia de pasos lógicos necesarios para llevar a cabo una tarea específica, como la solución de un problema.
- **Árbol binario**: Un árbol binario es aquel es el que cada elemento tiene máximo otros 2 elementos, comúnmente llamados hijo izquierdo e hijo derecho.
- **Árbol de búsqueda Binaria (ABB):** Es un árbol binario en el cual los elementos mayores a él, se ubican en su rama derecha, mientras que los elementos menores van en su rama izquierda. Cada elemento se almacena una sola vez por lo que no existen elementos repetidos.



- **Argumentos:** Los argumentos corresponden a la definición de los valores con los que la función va a operar.
- **Booleano:** Tipo de dato cuyo valor posible es True o False.
- **Cálculo lambda:** Formalismo que permite estudiar las propiedades de las funciones, facilitando la rigurosa evaluación y prueba de expresiones a través de una regla de transformación simple (sustituir variables) y definición de funciones.
- **Codificar:** Representación de una situación del mundo real a un conjunto de instrucciones que se puedan ejecutar por una computadora.

- **Código de máquina:** Es el único que entiende directamente la computadora, utiliza el alfabeto binario que consta de los dos únicos símbolos 0 y 1, denominados bits (abreviatura inglesa de dígitos binarios). Fue el primer lenguaje utilizado en la Programación de computadoras, pero dejó de utilizarse por su dificultad y complicación, siendo sustituido por otros lenguajes más fáciles de aprender y utilizar, que además reducen la posibilidad de cometer errores.
- **Código fuente:** Conjunto de líneas de texto escritas en un lenguaje de programación y que contienen las instrucciones que conforman la lógica del programa a ejecutarse en una computadora.
- **Diagramas de flujo:** Representación gráfica de un algoritmo o proceso.
- **Función:** Una función es un bloque de instrucciones definidas con el objetivo de llevar a cabo una labor específica. En el cuerpo de la función se presenta normalmente la forma de interacción de los parámetros (entrada), los procesos definidos y las relaciones necesarias entre estos para llegar al resultado esperado (salida).
- **Inducción matemática:** Método que permite probar o establecer que una propiedad se cumple no solo para un número finito de casos particulares, sino para una infinidad de ellos.
- **Instrucción:** Paso y/o acción que se define para estructurar el modelamiento del programa y que representa una operación a realizarse dentro de la ejecución del programa.
- **Iteración:** Repetición de un proceso o de un segmento de código.

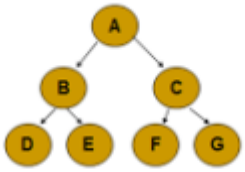
- **Lenguaje de alto nivel:** Se utilizan palabras de muy fácil comprensión para el programador. En contraposición, los lenguajes de bajo nivel son aquellos que están más cerca del "entendimiento" de la máquina. Algunos lenguajes de alto nivel son: Fortran, Cobol, C, Pascal, Ada, Basic, etc.
- **Lenguaje de programación:** Es un lenguaje formal que especifica una serie de instrucciones para que una computadora produzca diversas clases de datos. Los lenguajes de programación pueden usarse para crear programas que pongan en práctica algoritmos específicos que controlen el comportamiento físico y lógico de una computadora.
- **Lenguaje ensamblador:** Lenguaje compuesto por un conjunto de instrucciones básicas para los computadores, microprocesadores, micro controladores y otros circuitos integrados programables.
- **Notación Infija:** Notación común de fórmulas aritméticas y lógicas en la cual se escriben los operadores entre los operandos en que están actuando. Por ejemplo 4 + 1 [10].
- **Notación Postfija:** Notación de fórmulas aritméticas y lógicas en la cual el operador ocupa la posición después de los operandos en que está actuando. Por ejemplo 4 1 +.
- **Notación Prefija:** Notación de fórmulas aritméticas y lógicas en la cual se escriben los operadores al lado izquierdo de los operandos en que están actuando. Por ejemplo + 4 1.

Se evalúa de izquierda a derecha hasta que se encuentre el primer operador seguido inmediatamente de un par de operandos.

Se evalúa la expresión binaria y el resultado se cambia como un nuevo operando. Se repite este proceso hasta que quede un solo resultado.

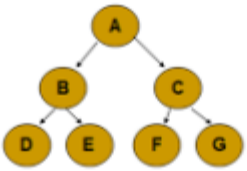
- **Operador:** Símbolo que actúa sobre los elementos a los cuales se le es asignado, cumpliendo una función predefinida. Por ejemplo: en la expresión a + b, el operador +, actúa sobre los elementos a y b, y su función es realizar la suma de dichos elementos. [11]
- **Operador aritmético:** Son aquellos operadores que actúan sobre datos numéricos. Dentro de este grupo se encuentra la suma (+), resta (-), multiplicación (*), división (/).
- **Operador lógico:** Símbolo o palabra que se utiliza para conectar dos fórmulas bien formadas o sentencias, de modo que el valor de verdad de la fórmula compuesta depende del valor de verdad de las fórmulas componentes. Los más comunes son AND, OR, NOT. AND genera un valor de verdad True solo cuando las expresiones evaluadas son verdaderas, OR genera un valor de verdad False sólo si las expresiones evaluadas son falsas, NOT niega el valor de verdad de la expresión evaluada, es decir, si la expresión es verdadera y le es incluido NOT su valor cambia a falso [12].
- **Operadores relacionales:** Permiten comparar expresiones que sean del mismo tipo, esta comparación es evaluada como una condición y su resultado dependerá de si se cumple o no el criterio de comparación. Operadores: <, >, >=, <=, ==. Ej 5>10 = False.

- **Paradigma:** Se refiere a la organización en el enfoque de pensamiento acordada o adoptada por una comunidad para tratar situaciones y/o solucionar problemas. En el caso de los paradigmas de programación, se trata de una formalización que se hace sobre la orientación en el razonamiento de las alternativas de solución que se deben de seguir en el tratamiento de un problema, por ejemplo, soluciones basadas en reglas lógicas (Programación lógica), a partir de la representación como objetos (Programación orientada a objetos), etc.
- **Proceso:** Programa en ejecución.
- **Programa:** Asociación de un conjunto de instrucciones ejecutables sobre una computadora y que realizan una tarea específica.
- **Programación declarativa:** Paradigma de programación en el que se especifican condiciones, afirmaciones, restricciones u otras declaraciones que permitan describir el problema a tratar y la alternativa de solución que han de usarse para el desarrollo de programas.
- **Programación funcional:** Paradigma de programación declarativo basado en el uso de las matemáticas para la elaboración de programas. Su base fundamental son las funciones, las cuales deben devolver el mismo resultado para los mismos argumentos, independientemente de la cantidad de veces que sea evaluada con ellos.
- **Pseudocódigo**: Es una forma de expresar el algoritmo utilizando el lenguaje natural, entendible para cualquier persona, pero añadiendo ciertas instrucciones típicas de los lenguajes de programación como lo son: condicionales, repetitivas.
- **Recorrido In-Orden:** Si se visita el primero hijo izquierdo, luego el padre y finalmente el hijo derecho.



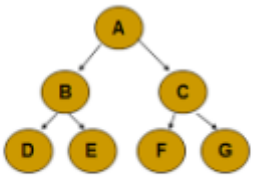
In-Orden: D-B-E-A-F-C-G

- **Recorrido Post-Orden:** Primero hijo izquierdo, luego el hijo derecho y finalmente el padre.



Post-Orden: D-E-B-F-G-C-A

- **Recorrido Pre-Orden:** Primero el padre, luego el hijo izquierdo y finalmente el hijo derecho.



Pre-Orden: A-B-D-E-C-F-G

- **Recursividad:** Propiedad que posee una función por la cual dicha función puede llamarse a sí misma.
- **Recursión directa:** Proceso mediante el cual una función, procedimiento o método se invoca a sí misma.

- **Recursión indirecta o mutua:** Proceso mediante el cual una función, procedimiento o método puede invocar a una segunda función, procedimiento o método que a su vez invoca a la primera.
- **String:** Tipo de dato compuesto por un conjunto de caracteres. Ejemplo: “Conceptos de programación”.
- **Lenguaje formal:** Sistema de comunicación estructurado definido para el uso en un contexto específico. Está constituido formalmente por símbolos primitivos (alfabeto) y reglas (gramática formal) para combinar estos símbolos.

3.3. MARCO REFERENCIAL

Diariamente las personas hacen uso de aplicaciones sin darse cuenta de cómo es su funcionamiento. Algunas de estas aplicaciones usan la programación funcional para llevar a cabo las actividades que normalmente las personas disfrutan. A continuación se habla de la aplicación más utiliza diariamente en la vida de la personas "WhatsApp". Esta aplicación ofrece servicios que demandan un alto nivel de concurrencia y que requieren de mucha disponibilidad y su infraestructura está desarrollada en Erlang (lenguaje funcional orientado a concurrencia). En el 2012, WhatsApp reportó que había logrado gestionar más de dos millones de conexiones por servidor, utilizando menos del 40 por ciento de la capacidad de procesamiento. Hoy en día WhatsApp fue adquirida por Facebook por más de 16 billones de dólares, cuenta con más de 1 billón de usuarios activos, procesa más de 42 billones de mensajes, 1.6 billones de fotos y 250 millones de videos al día. De esto se deduce la gran capacidad que puede aportar el paradigma de programación funcional en el desarrollo de soluciones efectivas para cumplir la demanda actual de información y servicios [13].

Existen otros casos en donde se puede apreciar como Scheme puede influir en el desarrollo de otras herramientas poderosas, algunos de ellos se listan a continuación [14].

- **Bigloo:** Es un sistema que permite programar en Scheme aplicaciones que generalmente requieren C. Se puede comunicar con C y Java [15].
- **Chicken:** Es un traductor de Scheme a C que se ejecuta en MacOS X, Windows y algunas versiones de Unix.
- **Elk:** Funciona como lenguaje de extensión para aplicaciones escritas en C y C++.

- **Galapagos:** Es un entorno de programación para Windows 95 basado en el intérprete SCM. Incorpora programación multihilo.
- **Guile:** Intérprete de Scheme distribuido en forma de biblioteca para ser incorporada dentro de un programa, posibilitando su extensión.
- **Gauche:** Intérprete de Scheme conforme al estándar R5RS que sirve como herramienta para el desarrollo de funciones diarias de los programadores. Funciona en varios sistemas tipo Unix.
- **Inlab-Scheme:** Es una implementación de Scheme conforme al estándar R4RS. Se emplea en el procesamiento de imágenes y reconocimiento de patrones. Se distribuye un ejecutable para Linux [16].
- **Jscheme:** Es un intérprete de Scheme en Java de acuerdo con el estándar R4RS salvo por tres limitaciones [17]. Sólo ocupa 30 Kb.
- **Kawa:** Es un sistema de Scheme basado en Java que produce bytecode.
- **KSM-Scheme:** Es un intérprete para Linux del estándar R5RS con extensiones para hacer uso de funciones y variables en C.
- **Larceny:** Es un sistema de ejecución Scheme simple y eficiente basado en un relativamente simple compilador optimizado llamado Twobit. Fue creado originalmente como un medio para la investigación sobre recolección de basura (garbage collection) y optimizaciones de compilación.
- **MIT/GNU Scheme:** Es un completo sistema de programación Scheme para Unix, Windows y OS/2.

- **Open Scheme:** Es un intérprete y compilador para Linux, Solaris, BeOS, FreeBSD y Windows. Hay versiones comerciales y de evaluación, gratuitas para usos no comerciales.
- **OScheme:** Es un pequeño intérprete incrustarle conforme con R4RS aunque con algunas restricciones y extensiones. Se distribuye sólo el código fuente (que se ha compilado en varios sistemas, incluyendo Intel-Linux.
- **Petite Chez Scheme:** Es una versión gratuita de Chez Scheme, un Scheme comercial de Cadence Research Systems. Hay versiones para Windows, Intel Linux, MacOS X y Sparc Solaris.
- **PHPScheme:** Es un intérprete de Scheme escrito en PHP.
- **PLT-Scheme:** Es un nombre que agrupa varias implementaciones de Scheme, siendo la principal DrScheme. DrScheme es un entorno gráfico interactivo de programación para Windows, MacOS X y Unix/X.
- **QScheme:** Es una implementación de Scheme pequeña y rápida escrita en C.
- **RScheme:** Es una implementación de Scheme con orientación a objetos adaptada del lenguaje Dylan. Puede producir código C o bytecode interpretable por una máquina virtual.
- **TinyScheme:** Es un intérprete que implementa un subconjunto del estándar R5RS. Está pensado para ser usado desde otros programas. Sólo se distribuye el código fuente.

4. PROPUESTA DE DOCUMENTO

En este capítulo se profundizará en el campo de la programación funcional cuya fundamentación está vinculada a la demostración lógico-matemática representada en el cálculo lambda y a partir de la cual se dará inicio.

- **Calculo lambda**

El cálculo lambda es un tipo de sistema lógico-deductivo que se compone de:

- Un lenguaje formal.
- Una gramática formal que permite validar y limitar las expresiones que se pueden formar con el lenguaje.
- Reglas de inferencia las cuales sirven para concluir a través de la generación de premisas y el análisis de su sintaxis.
- Axiomas para hacer derivaciones proposicionales.

Su importancia radica en que permite definir funciones formalmente y una de las principales características de los lenguajes funcionales es el uso de funciones para la elaboración de programas.

En el proceso de definición empleado por el cálculo lambda no se da nombre a las funciones [18], se utiliza el símbolo lambda (λ) en su definición. Además se tiene que una **abstracción** es una **función**, que corresponde a un término lambda con una cabeza y un cuerpo y que se aplica a un argumento que se conoce como el valor de entrada o input.

El cálculo lambda permite efectuar dos operaciones:

- Abstracción funcional: Consiste en la definición de funciones de un solo argumento y un cuerpo específico, y se denota λx.B.
- Aplicación de función: Consiste en aplicar las funciones definidas sobre un argumento real (A). Denotado de la siguiente forma (λx.B) A.

Un ejemplo de lo anterior es el siguiente [19]:

Tabla 1Tabla 1Desarrollo de una función simple empleando cálculo lambda

	Función	**Evaluación como función**	**Cálculo Lambda**	**Evaluación con cálculo lambda**
Definición formal	Suma(x) = x + 1		Λx. x + 1	
Cabeza	Suma(x)	Evaluar con 3. Suma(x) = x + 1 Suma(3) = 3 + 1 Suma(3) = 4	Λx	Evaluar con 3. Λx. x + 1 (Λx. x + 1)3 (x := 3) (3 + 1) 4
Nombre	Suma		No existen nombres	
Parámetro	"(x)", encontrado dentro de Suma(x)		x, ubicado a la derecha de λ	
Cuerpo	x + 1		x + 1	
Signo	"=", separa al cuerpo de la cabeza.		"." Separa al cuerpo de la cabeza	

En el cálculo lambda las abstracciones tienen variables que pueden ser **ligadas o libres**. Una variable es **ligada** si se encuentra definida en la función y también en el cuerpo de la misma [lamb2], por ejemplo:

En la abstracción **Λx.xy:**

- x es una variable ligada porque se encuentra en el cuerpo de la abstracción.
- y es una variable libre porque no se utiliza.

Las abstracciones reciben solo un valor de entrada, debido a esto, cuando una función tenga 2 o más parámetros de entrada y se quiera representar haciendo uso del cálculo lambda será necesario escribir una abstracción que retorne otra. Por ejemplo:

Tabla 2Desarrollo de una función compuesta empleando cálculo lambda

Función	**Cálculo lambda**
f(x,y) = 2x + 3y Evaluando con 5 y 8 f(5, 8) = 2(5) + 3(8) f(5, 8) = 10 + 24 f(5, 8) = 34	Λxy.2x + 3y Λx.Λy.2x + 3y Λx.(Λy.2x + 3y) Evaluando con 5 y 8 (Λxy.2x + 3y) 5 8 (Λx.(Λy.2x + 3y)) 5 8 [x := 5] (Λy.2(5) + 3y) 8 (Λy.10 + 3y) 8 [y := 8] (10 + 3(8)) (10 + 24) 34

De esta manera se emplea el cálculo lambda en el proceso de construcción y evaluación de funciones. Ahora bien, para emplear la programación funcional en la elaboración de programas se requiere pensar soluciones en términos de funciones, diseñar el algoritmo que dará solución al problema y plasmar este algoritmo en el lenguaje de programación funcional que se haya definido para implementarlo. Para ello, primero se debe saber que es un algoritmo:

Algoritmo

"Un algoritmo se puede definir como una secuencia de instrucciones que representan un modelo de solución para determinado problema. [20]".

Para cada problema el algoritmo se puede expresar en un lenguaje de programación diferente y ejecutarse en una computadora distinta; sin embargo el algoritmo será siempre el mismo. Esto se puede ilustrar más claramente de la siguiente manera: una receta de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el idioma, los pasos (algoritmo) para la elaboración del plato se realizarán sin importar el cocinero (computadora). Por ejemplo:

- Los pasos matemáticos para la solución de un número factorial.
- Los pasos matemáticos para hallar el área de un triángulo.
- Los pasos para realizar una llamada telefónica.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar definido. Si se siguen los pasos de un algoritmo varias veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea debe de tener un número finito de pasos.
- La definición de un algoritmo debe describir tres partes: Entrada, Proceso y Salida.

En el algoritmo mencionado anteriormente se tendrá:

- **Entrada:** Ingredientes y utensilios empleados.
- **Proceso:** Elaboración de la receta de cocina.
- **Salida:** Terminación del plato (por ejemplo: Salmón a la plancha)

Un algoritmo exige que se tengan varias propiedades importantes:

Los pasos de un algoritmo deben ser simples y exentos de ambigüedades (diferentes significados), deben seguir un orden cuidadosamente prescrito, deben ser efectivos y deben de resolver el problema en un número finito de pasos.

El siguiente ejemplo muestra un algoritmo para cambiar un bombillo quemado.

Cambiar un bombillo quemado podría resumirse en dos pasos:

- Quitar el bombillo quemado.
- Colocar un bombillo nuevo.

Pero, si se quiere entrenar a un robot para que efectúe esta tarea, se debe especificar y describir detalladamente los pasos a seguir, por ejemplo, suponga que el bombillo se encuentra en el techo de la cocina, al momento de especificar cada una de las acciones a realizar se tendría:

- Situar la escalera debajo del bombillo quemado.
- Elegir el bombillo de reemplazo (de las mismas características que el anterior).
- Subir por la escalera hasta alcanzar el bombillo.
- Girar el bombillo contra las manecillas del reloj hasta que esté suelto.
- Ubicar el bombillo nuevo en el mismo lugar que el anterior.
- Enroscar el bombillo en el sentido de las manecillas del reloj hasta que quede ajustado.
- Bajar de la escalera.

En la actualidad existen diferentes mecanismos para representar algoritmos, se ha adoptado el pseudocódigo y los diagramas de flujo.

Pseudocódigo

Es un lenguaje algorítmico más amigable que un lenguaje de programación, que permite centrar la atención en la resolución del problema y no en los detalles propios de un lenguaje de programación [21]. El pseudocódigo es una notación algorítmica textual caracterizada por:

- Permitir la declaración de los datos manipulados por el algoritmo.
- Apela a las normas de estructura de un lenguaje de programación aunque está pensado para que pueda ser leído por un ser humano y no interpretado por una máquina.

Diagrama de flujo

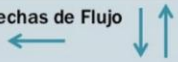
Es una manera de representar gráficamente un algoritmo o proceso, dicho de otra manera, representa la secuencia o pasos lógicos y ordenados para realizar una tarea mediante el uso de símbolos y descripciones textuales del procedimiento que se esté desarrollando. Los diagramas de flujo describen que operaciones y en qué secuencia se requieren realizar para buscar una solución [22].

- Los diagramas de flujo facilitan la comunicación entre el programador y el resto de las personas.
- Permite describir de manera más entendible el funcionamiento de un algoritmo.
- Facilita la codificación del programa en cualquier lenguaje de programación.

Algunas de las herramientas que facilitan la construcción de diagramas de flujo son:

- **www.draw.io**: Permite guardar los diagramas en el disco duro de tu ordenador, Gdrive, OneDrive o Dropbox.
- **www.lucidchart.com**: Posibilita crear diagramas de flujo o mapas mentales.

A continuación se muestran una serie de símbolos útiles para llevar a cabo este tipo de representaciones.

símbolo	Función	Símbolo	Función
Terminal 	Indicar el inicio y fin del diagrama	Teclado 	Introducir datos manualmente por el teclado
Entrada/salida 	Entrada o salida simple de información	Decisión 	Indica operaciones lógicas o de comparación y tienen dos salidas dependiendo del resultado.
Proceso 	Realizar cualquier operación o calculo con la información		
Salida a Impresora 	Salida de informacion a la impresora	Conectores 	Une dos partes del diagrama a la misma o diferente página
Salida a Pantalla 	Mostrar información de salida a la pantalla	Flechas de Flujo 	Indica la direccion del flujo de la información

Los métodos descritos anteriormente permiten definir y describir el mecanismo de solución a emplear en la resolución de un problema. No obstante, es de vital importancia pensar en la definición del algoritmo en términos de funciones si se habla de programación funcional.

No se debe olvidar que el algoritmo es la infraestructura de cualquier solución y las funciones son el eje de la programación funcional.

Se ha relacionado en múltiples oportunidades la palabra “función”, pero no se ha profundizado en su uso, para hacerlo, primero se va a definir un lenguaje de programación para ejemplificar algunos de los conceptos que se van a ver. En este caso se hará uso de Scheme, a continuación se presentan algunas de sus ventajas [5].

Scheme

- Su sintaxis es muy simple y regular. La sintaxis de Scheme emplea notación prefija y hace uso de paréntesis para agrupar cada conjunto de datos y las operaciones que los involucran.
- Es un lenguaje orientado a las expresiones caso contrario a la mayoría de los lenguajes que se orientan a sentencias o instrucciones; lo que implica que tienen que diferenciar las partes del código que se comportan como una expresión matemática, de las que cambian el estado de un programa. Scheme hace que las expresiones realmente se comporten como expresiones matemáticas y sean indistinguibles de las sentencias.
- No requiere que se declaren variables y toma como tipo la propiedad inherente a los objetos y no a las variables que los nombran [23].
- El mecanismo de paso de parámetros es por valor, es decir, se pasa el valor en tiempo real y no la dirección de la posición de memoria donde el parámetro real está almacenado (paso por referencia). Esto le permite al programador tener más control sobre los efectos que se puedan producir durante la ejecución del programa.
- Las funciones pueden acceder únicamente a las asociaciones de un valor con un objeto que formen parte de sus vinculaciones locales y de las que contenga textualmente la función.
- Se puede implementar el mecanismo de iteración eficientemente haciendo uso de la recursión y no de los ciclos que normalmente se utilizan en los otros lenguajes.
- Soporta diferentes paradigmas de programación como el funcional, el imperativo y el orientado a objetos.

- Permite la evaluación concisa a través del uso de streams que son una secuencia de objetos similar a las listas o los vectores y que permiten la evaluación de sus componentes cuando se accesan y no cuando se almacenan, lo que permite representar estructura de datos de gran tamaño en espacio limitado.
- Permite la realización de programas basados en datos simple como enteros, strings booleanos.

Lo mencionado anteriormente hace de Scheme una herramienta poderosa y sencilla para abordar los conceptos básicos de la programación funcional. Una vez mencionados aspectos importantes de este lenguaje se procederá a continuar con las implicaciones del uso de funciones.

Función

Una función es una definición de un procedimiento que describe la relación entre una entrada y una salida y es caracterizada por contar con los siguientes elementos:

a) **Objetivo:** Atribuye el propósito para el cual se va a crear dicha función. La función debe alcanzar el objetivo que lo define y aportar a la necesidad del problema que se quiere resolver. El objetivo es lo que realmente hace práctico crear la función.
b) **Nombre:** En esencia, el nombre pretende darle una identificación propia a la función para que ésta pueda ser llamada en cualquier momento.
c) **Argumentos:** Son los valores que hacen que la función opere. Es equivalente a los ingredientes necesarios para preparar una receta. La cantidad de argumentos varía de acuerdo a la definición de la solución para cada problema.
d) **Validaciones:** Son las restricciones que deben verificarse en relación con los argumentos que han de alimentar la función.

e) **Proceso:** Es la definición del procedimiento como tal, dicho de otra manera, corresponde a la descripción de las formas de interacción de los argumentos que se han establecido y que está asociada a la lógica que permite que la función alcance el objetivo para la cual fue creada.

f) **Resultado:** Es el dato resultante de la ejecución de la función con los parámetros definidos. Resulta de la interacción de los argumentos y el proceso para ello. El resultado siempre debe ser el mismo cuando la función se ejecuta con los mismos argumentos.

Esta es una de las razones principales para que las funciones sean lo más importante cuando se habla del uso de este paradigma para la resolución de problemas, debido a que en paradigmas como el imperativo, el resultado obtenido puede variar aun si se ejecutan las soluciones con los mismos argumentos.

- **Composición de funciones**

Las funciones pueden ser tan simples como una suma o tan complejas como aquellas en donde se asocian varias expresiones aritméticas y relaciones.

Una función en Scheme se forma a partir de la articulación de datos y funciones primitivas relacionadas a través de una lógica de programación. Algunos de los datos con los que trabaja Scheme son los siguientes [25]:

- **Booleanos**

Tipo de dato lógico que representa un valor de verdad verdadero o falso. En el lenguaje Scheme son representados con la siguiente notación: **#t (verdadero)** y **#f (falso).** Por ejemplo:

#t; verdadero

#f; falso

(> 5 2)

- **Números**

Scheme soporta gran variedad de tipos de datos numéricos incluyendo racionales, complejos e inexactos. Ejemplo: 42, 3.6, 2+3i

- **Caracteres**

Se representan de la siguiente manera en lenguaje Scheme **#\a, #\b,** donde a y b son los caracteres representados. Se soportan caracteres internacionales y se codifican en UTF-8.

- **Cadenas**

Las cadenas son consideradas como una composición de caracteres o secuencias finitas de estos. Se escriben entre comillas dobles.

Ejemplo: “Hola”, “Programacion funcional”, “Racket”

- **Simbolos**

Son los denominados identificadores en otros lenguajes de programación. Corresponden a objetos simples y no puede contener espacios, características que los diferencian de los strings. Ejemplo: ´Hola, ´Mansion

- **Listas**

Son colecciones de elementos homogéneos. Se representan y crean de la siguiente manera:

(list 1 2 3 4 ; list crea una lista

'(1 2 3 4)

Existen otros tipos de datos, pero los mencionados anteriormente constituyen la base para la creación de funciones.

A continuación, se mostraran algunas de las funciones primitivas que se encontraran en Scheme, los tipos de datos que reciben y cuales son los datos de salida.

Función	Tipo de entrada	Salida	Descripción
+	num, num,…	num	Suma.
-	num,num,…	num	Resta.
/	num,num,…	num	División.
>	real, real, ...	bool	Compara si el primer valor es mayor a los demás.
<	real, real, ...	bool	Compara si el primer valor es menor a los demás.
>=	real, real, ...	bool	Compara si el primer valor es mayor o igual a los demás.
<=	real, real, ...	bool	Compara si el primer valor es menor o igual a los demás.
complex?	any	bool	Evalúa si algo es o no un número complejo.
cos	num	num	Coseno.
acos	num	num	Arco-seno.
expt	num,num	num	Eleva un número a una determinada potencia.
log	num	num	Calcula el logaritmo de un número.
max	real, real, ...	num	Evalúa uno o más números y retorna el mayor.
min	real, real, ...	num	Evalúa uno o más números y retorna el menor.
sqrt	num	num	Calcula la raíz cuadrada de un número.
round	real	int	Redondea un número.

- **Definición de una función**

La sintaxis para definir un procedimiento es la siguiente:

(define (nombre_funcion argumentos) cuerpo de la función)

Por ejemplo, si se quiere realizar una función simple que eleve un número al cuadrado se tendría:

(define (cuadrado p)

(* p p))

Para elevar el número 3 al cuadrado haciendo uso de esa función se debe ejecutar ***(cuadrado 3)*** y como resultado se obtendría el valor de 9.

Un ejemplo en el que se puede ver el uso de funciones también se da en la resolución del siguiente problema:

Un hombre camina 5 km por hora a paso normal. Escribir una función que permita calcular cuántas horas le tomará caminar una distancia determinada.

La solución a este problema es sencilla, distancia en kilómetros / 5 km = horas necesarias. Haciendo uso de la notación de función en el lenguaje Scheme se tiene:

(define (tiempoCamina distancia)

distancia / 5

)

Se ha definido una **función** llamada **tiempoCamina,** que recibe como **argumento o valor de entrada** la distancia a la que se quieres calcular el tiempo que demorará recorrerla, por último se tienen las **instrucciones** a realizar con la función y que corresponde a dividir la distancia entre cinco para conocer el tiempo que tomará recorrer la distancia ingresada. Para hacer uso de esta función basta con ejecutar:

(tiempoCamina x) donde x es la distancia en número. Ejemplo

(tiempoCamina 500) = 100

Beneficios de la Programación Funcional.

- **Pruebas unitarias.**

Ninguna función puede causar efectos colaterales. No puede modificar los símbolos que ya tienen un valor, y ninguna función puede modificar un valor fuera de su ámbito para ser usado por otra función. Eso significa que el único efecto de evaluar una función es su valor de retorno y que la única cosa que afecta el valor de retorno de la función son sus argumentos. Se puede probar cada función del programa preocupándose únicamente de sus argumentos [27].

- **Depuración.**

En un programa funcional no funciona como se espera, depurarlo es cosa fácil. El problema se repetirá siempre, pues un programa funcional no depende de lo que ocurra antes o después en otra parte que no sea la función misma. En un programa imperativo, un problema puede aparecer algunas veces y otras no. Dado que las funciones en un programa imperativo dependen de estados externos producidos por efectos colaterales de otras funciones.

Una vez identificado el problema se detiene la ejecución del programa y examinas la pila. Cada argumento en la llamada a una función está en la pila disponible para su inspección, igual que en un programa imperativo. Excepto que en un programa imperativo esto no es suficiente, pues además hay que revisar variables miembros, variables globales y el estado de otras clases.

- **Concurrencia.**

No necesita preocuparse por condiciones de ejecución o bloqueos mutuos, la razón es porque no usa bloqueos, ninguna pieza de datos en un programa funcional es modificada dos veces en el mismo hilo de ejecución, mucho menos en hilos diferentes. Esto significa que puedes agregar hilos a tu aplicación, libre de los problemas que plagan a las aplicaciones concurrentes en los programas imperativos.

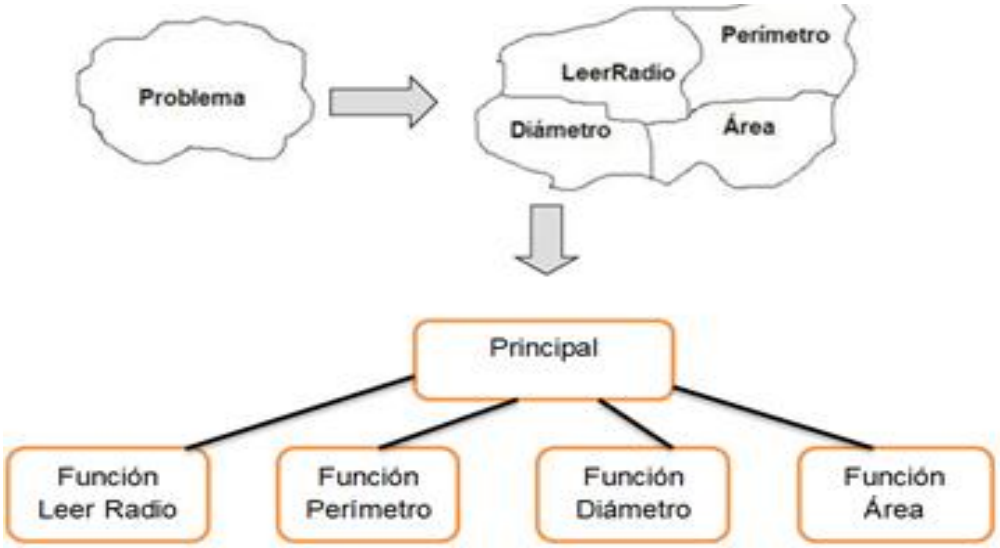
También es necesario considerar que la programación concurrente permite que los sistemas sean más fácilmente escalables debido a la modularidad de su desarrollo y que también estos pueden ser mucho más eficientes debido a que permiten la ejecución en paralelo de múltiples instrucciones. La simulación es una herramienta importante en la optimización de sistemas físicos; la programación concurrente brinda una forma natural de asignar segmentos del programa para representar objetos físicos y por eso ayuda mucho a representar simulaciones.

- **Modularidad**

La modularidad consiste en que un programa se construya a partir de la unión entre funciones de manera que cada función sea independiente pero a la vez interdependiente, es decir, que cada función cumpla con un objetivo específico pero que en conjunto puedan cumplir con un objetivo mayor.

Observe el siguiente ejemplo:

Calcular el área, perímetro y diámetro de un círculo de radio R.



Una de los recursos importantes que se tienen dentro de la programación funcional es la recursividad, que consiste en usar la misma función que se define dentro de su propia definición. Normalmente, la forma en que son llamadas, se da dependiendo del resultado de una prueba que permite establecer en qué lugar debe estar y de que debe estar acompañado el llamado para lograr el resultado esperado. El uso de llamados de funciones sobre si mismas exige que deba establecerse una condición de parada, es decir, algo que impida que se ejecute la función infinitamente.

Problemas implícitos en la programación concurrente.

Al programar concurrentemente y esto conlleva a compartir recursos, surgen algunos problemas que necesitan ser resueltos para así aprovechar al máximo todas las ventajas que la programación concurrente puede ofrecer.

Entre los problemas más importantes se puede mencionar:

- La ejecución de un proceso que puedan afectar la información perteneciente a otro proceso que se ejecuta en paralelo (datos compartidos).
- **Bloqueo mutuo:** Es el estado en el que dos transacciones se queden bloqueadas cada una esperando por recursos que está utilizando la otra.
- **Inanición:** Es donde a un proceso se le deniega siempre el acceso a un recurso compartido.
- **Livelock:** Estado en donde una transacción cambia continuamente de estado en respuesta a cambios en otra transacción mientras la otra hace lo mismo, sin conseguir ningún resultado con ello.
- **Actualizaciones en caliente (Hot Code Deployment)**

En las primeras versiones para instalar actualizaciones de Windows era necesario reiniciar la computadora varias veces. Y eso solo para instalar una nueva versión del reproductor multimedia. Con Windows XP la situación había mejorado bastante, pero aun le faltaba para que fuera ideal. Los sistemas Unix han tenido un mejor modelo desde hace tiempo. Para instalar una actualización, solo necesitas detener los componentes relevantes, no el sistema operativo entero. Los sistemas de telecomunicaciones necesitan estar al 100% todo el tiempo, pues si por una actualización del sistema no se puede realizar una llamada de emergencia algunas vidas podrían perderse.

Una situación ideal sería actualizar las partes importantes del código sin reiniciar ni detener ninguna parte del sistema. En un mundo imperativo, eso sería imposible. Por ejemplo para el cambio de una clase en Java, se tendría que realizar grandes cambios a nivel lógico y estructural del programa para que este funcione adecuadamente.

En un programa funcional todo el estado está guardado en la pila, en los argumentos pasados a las funciones. Esto hace que las actualizaciones en caliente sean significativamente más fáciles. El resto puede ser hecho por las herramientas de lenguaje, automáticamente. Los ingenieros de Erlang han estado actualizando sistemas sin detenerlos.

Pruebas y optimizaciones asistidas por computadora

Una característica interesante de los lenguajes funcionales es que se puede razonar sobre ellos matemáticamente. Todas las operaciones matemáticas que pueden hacerse sobre papel también aplicarán a los programas escritos en dicho lenguaje. El compilador puede, por ejemplo, convertir piezas de código en equivalentes más eficientes con la comprobación matemática de que ambas piezas de código son equivalentes (Lo contrario no siempre es verdad. Mientras que a veces es posible probar que dos piezas de código son equivalentes, esto no es posible en todas las situaciones). Las bases de datos relacionales han usado este tipo de optimizaciones por años. No hay razón por la que estas mismas técnicas no se usen en otro tipo de software.

Adicionalmente, puedes usar estas técnicas para probar que ciertas partes de tu programa son correctas. ¡Hasta es posible crear herramientas que analicen el código y generen casos de pruebas para comprobación de unidades automáticamente! Esta funcionalidad es invaluable para sistemas sólidos como las rocas. Si estás diseñando marcapasos o sistemas de control de tráfico aéreo estas herramientas son casi siempre un requerimiento. Si estás escribiendo aplicaciones que no pertenezcan a las verdaderas industrias de misión crítica, de todas formas estas herramientas te darán una tremenda ventaja sobre tus competidores.

Evaluación tardía

La evaluación tardía (o perezosa) es una técnica interesante que se vuelve posible una vez que se adopta una filosofía funcional:

```
1    String p1 = OperacionMuyLarga1();
2    String p2 = OperacionMuyLarga2();
3    String p3 = concatenar(p1, p2);
```

En un lenguaje imperativo el orden de evaluación queda claro. Dado que cada función pudiera afectar o depender del estado externo, es necesario ejecutarlas en orden: primero *OperacionMuyLarga1*, luego *OperacionMuyLarga2*, seguida de *concatenar*. Caso que no pasa en un lenguaje funcional.

Como se vió antes, *OperacionMuyLarga1* y *OperacionMuyLarga2* pueden ser ejecutadas concurrentemente porque está garantizado que ninguna función afecta o depende del estado global. Solo se necesita ejecutar estas operaciones cuando otra función dependa de *p1* y *p2*. Ni siquiera se debe ejecutar antes de *concatenar*. Si se reemplaza *concatenar* con una función que tenga una condición y use sólo uno de sus parámetros, Haskell es un ejemplo de lenguaje de evaluación tardía. En Haskell no hay garantía de que el código sea ejecutado en orden, pues Haskell solo ejecuta el código cuando es requerido.

La evaluación tardía tiene numerosas ventajas así como desventajas. A continuación se mencionaran algunas de las ventajas y las desventajas.

Ventajas

- **Optimización:** La evaluación tardía ofrece un tremendo potencial para optimizaciones. Un compilador de este tipo ve al código funcional exactamente como un matemático ve una expresión algebraica. Puede cancelar cosas y prevenir completamente su ejecución si es necesario, reacomodar las piezas para mayor eficiencia, incluso reacomodar el código de una forma que reduzca los errores, todo esto garantizando que la lógica permanecerá intacta. Este es el mayor beneficio de representar programas usando estrictamente primitivas formales: como el código se adhiere a leyes matemáticas, se puede pensar en él matemáticamente.
- **Abstracción de estructuras de control:** La evaluación tardía provee un nivel superior de abstracción que permite implementar cosas que de otra forma serían imposibles.
- **Estructuras de datos infinitas:** Los lenguajes de evaluación tardía permiten la definición de estructuras de datos infinitas. Por ejemplo, considera una lista con los números de Fibonacci. Está claro que no se puede realizar cálculos sobre una lista infinita en un tiempo razonable, o guardarla en memoria. En lenguajes imperativos como Java, simplemente se define una función Fibonacci que devuelva un número particular de la secuencia. En un lenguaje como Haskell se puede abstraer aún más y simplemente definir una lista infinita con los números de Fibonacci. Como el lenguaje es de evaluación tardía, sólo las partes necesarias de la lista que son usadas realmente por el programa serán evaluadas.

Desventajas

La principal desventaja es que es tardía (o perezosa). Muchos de los problemas del mundo real requieren evaluación estricta. Por ejemplo:

```
1.System.out.println("Por favor ingresa tu nombre: ");
2.System.in.readLine();
```

En un lenguaje de evaluación imperativa no hay garantía de que la primera línea se ejecutará antes de la segunda, esto conlleva a que no se pueda hacer operaciones de Entrada/Salida. Los matemáticos han trabajado y desarrollado algunos trucos para asegurarse de que porciones del código sean ejecutadas en un orden particular en un ambiente funcional. Estas técnicas incluyen continuaciones, monads, y escritura singular. En esta guía solo se veran las continuaciones.

Continuaciones

Cuando se estudian las funciones, se asume falsamente que estas deben regresar un valor de retorno a donde se hizo el llamado. En este sentido, las continuaciones son una generalización de las funciones. Una función no necesita regresar a quien la llamó, y más bien puede ir a cualquier otra parte del programa. Una “continuación” es un parámetro opcional que le dice a la función donde debe continuar la ejecución del programa. Mira el fragmento de código del listado siguiente:

```
1.int i = suma(20, 10);
2.int j = cuadrado(i);
```

La función suma devuelve 30, valor que es asignado al símbolo i, en el lugar donde suma fue llamado. Después el valor de i es usado para llamar cuadrado. Note que un compilador de evaluación imperativa no puede reacomodar estas líneas de código pues la segunda línea depende de la evaluación correcta de la primera. Se puede reescribir este bloque de código usando Continuación o CPS (Continuation Pass Style), donde la función suma no regresa a quien la llamó sino entrega su resultado a la función cuadrado.

```
1.int j = suma(20, 10, cuadrado);
```

En este caso a suma se le pasa otro parámetro, una función a la que suma debe llamar cuando haya terminado su trabajo. En este caso, cuadrado es la continuación de suma. En ambos casos j será igual a 900.

Así que los programas en estilo CPS no necesitan pila sino un argumento extra con la función a la cual llamar a continuación. Los programas que no están escritos en estilo CPS no tiene el argumento de continuación extra, sino la pila. La pila simplemente contiene los argumentos y un puntero a donde la función debe retornar. La pila contiene únicamente información de continuación, el puntero para la instrucción de retorno en la pila es esencialmente lo mismo que la función a llamar en programas CPS.

Cuando se obtiene la continuación actual y se guarda en algún lugar, en realidad se está guardando el estado actual de nuestro programa (como si se congelara en el tiempo). Esto es similar a cuando el sistema operativo entra en hibernación. Un objeto de continuación contiene la información necesaria para reiniciar el programa desde el punto donde fue creado. Un sistema operativo hace esto todo el tiempo cuando cambia de contexto. La única diferencia es que el sistema operativo lo controla todo. Si tu solicitas un objeto de continuación (en el lenguaje Scheme se hace mediante llamar a la función call-with-current-continuation) obtienes un objeto que contiene la continuación actual (toda la información en la pila, o la siguiente función a ser llamada en el caso de CPS). Ahora puedes guardar este objeto en una variable. Cuando deseas "reiniciar" tu programa con este objeto de continuación, "transformas" el estado de tu programa al que está guardado en el objeto de continuación.

Coincidencia de patrones (Pattern Matching)

La coincidencia de patrones no es una característica innovadora. De hecho, tiene poco que ver con la programación funcional. La única razón por la que usualmente se le atribuye es porque los lenguajes funcionales han tenido coincidencia de patrones por algún tiempo, mientras que los lenguajes imperativos modernos aún no.

Observe la función Fibonacci en el siguiente ejemplo:

```
int fib(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
      return fib(n-2) + fib(n-1);
  }
```

El listado siguiente muestra la función Fibonacci en nuestro lenguaje derivado de Java con soporte para coincidencia de patrones.

```
int fib(0){
```

```
return 1;
}
int fib(1){
return 1;
}
int fib(int n){
return fib(n-2) + fib(n-1);
}
```

Como se puede observar, aumentaron el número de funciones y sentencias swicth complicadas, en la programación funcional se decidió que sería una buena idea abstraerse de esta forma. Cuando la función es llamada, el compilador compara los argumentos con las definiciones en tiempo de ejecución y elige la correcta. Esto se hace usualmente eligiendo la definición más específica. Por ejemplo, int fib(int n) podría ser llamada siendo n igual a 1, pero en este caso int fib(1) es más específica.

La coincidencia de patrones es usualmente más compleja de la que se observa. Por ejemplo, un sistema avanzado de coincidencia de patrones nos permitiría hacer lo siguiente:

```
1.int f(int n &lt; 10){ ... } int f(int n) { ... }
```

Es útil la coincidencia de patrones cada vez que se tiene una compleja estructura de ifs anidados, se puede hacer un mejor trabajo con menos código. Otro beneficio de la coincidencia de patrones es que si se necesita añadir o modificar condiciones, no se tiene que lidiar con una única función enorme. Simplemente se añade (o modificas) las definiciones apropiadas.

5. CONCLUSIONES

Los enfoques usados para darle solución a problemas son variados, pero las características que provee la programación funcional permiten obtener soluciones a problemas complejos sin obtener variaciones cada vez que se pruebe el mismo caso, cosa que en ocasiones se presentan cuando se emplean otros tipos de paradigmas en la solución. No obstante, a pesar de que ofrece algunas ventajas, el proceso de difusión e implementación a escala de esta alternativa de solución de problemas requiere un mayor tiempo debido a que su implementación está actualmente enfocada en el ámbito educativo e investigativo y se requiere ampliar su incidencia en el campo comercial.

En contraste con otros paradigmas de programación, este permite la especialización en el proceso de construcción de soluciones debido a que su lógica está orientada a la elaboración de funciones cuya lógica pueda operar con los diferentes casos posibles y brinde seguridad en los resultados del proceso. Para ello se necesita comprender y estructurar el esquema del problema de manera que se pueda obtener una interpretación computacionalmente viable, de la que se pueda obtener resultados en un número finito de pasos y aprovechar la mayor cantidad de recursos ofrecidos por el equipo en donde se ejecutará el programa.

6. BIBLIOGRAFÍA

[1] César Vaca Rodríguez, D. d. (11 de Febrero de 2011). Obtenido de Escuela de Ingeniería Informática de Valladolid: https://www.infor.uva.es/~cvaca/asigs/docpar/intro.pdf

[2] Trejos-Buriticá, O. I. (25 de Noviembre de 2015). *Revista Educación en Ingeniería.* Obtenido de https://www.educacioneningenieria.org/index.php/edi/article/download/719/315

[3] O'Regan, G. (2013). *Giants of Computing.* London: Springer-Verlag .

[4] Lips-Monografia, Obtenido de https://www.scribd.com/document/179110624/LISP-Monografia

[5] Coello Coello, Carlos A. "Scheme Lo pequeño es bello (Primera Parte*)", Soluciones Avanzadas. Tecnologías de Información y Estrategias de Negocios*. Año 4, No. 39, 15 de noviembre de 1996, pp. 27-34.

[6] Erlang, https://www.ecured.cu/Erlang

[7] Haskell*, https://www.ecured.cu/Lenguaje_de_programaci%C3%B3n_Haskell*

[8] Scala. (s.f.). *Scala*. Obtenido de https://docs.scala-lang.org/es/tutorials/tour/tour-of-scala.html.html

[9] Jones, M. T. (8 de Marzo de 2018). *ibm.com*. Obtenido de https://www.ibm.com/developerworks/ssa/library/os-developers-know-rust/index.html

[10] Cristóbal, S. (s.f.). *www.monografias.com.* Obtenido de https://www.monografias.com/trabajos44/pilas-listas-expresiones/pilas-listas-expresiones2.shtml

[11] *luzumisu*. (24 de Febrero de 2009). Obtenido de http://luzumisu.over-blog.com/article-28322968.html

[12] ecured. (s.f.). *ecured*. Obtenido de https://www.ecured.cu/Operadores_lógicos

[13] Importancia de la Programación Funcional en un Mundo Paralelo, https://medium.com/@Loopa/importancia-de-la-programaci%C3%B3n-funcional-en-un-mundo-paralelo-45d8425f6047

[14] Scheme- Implementaciones, *http://www.rodoval.com/paginalen.php?len=Scheme*

[15] Bigloo, *http://www-sop.inria.fr/mimosa/fp/Bigloo/*

[16] Inlab-Scheme, https://www.inlab.net/inlab-scheme/

[17] Jscheme, *http://www.norvig.com/jscheme.html*

[18] Ordonez, M. (23 de Octubre de 2016). *mordonez.me*. Obtenido de http://www.mordonez.me/un-primer-vistazo-al-calculo-lambda/

[19] rayskell (3 de Abril de 2017). *rayskell.com*. Obtenido de https://www.rayskell.com/lambda-calculo

[20] *Universidad Nacional del Noreste.* (s.f.). Obtenido de http://ing.unne.edu.ar/pub/informatica/Alg_diag.pdf

[21] aguilar, L. j. (2008). *Fundamentos de programación.* Mcgraw-hill / interamericana de españa, s.a.

[22] *Representación de un algoritmo*. Obtenido de http://formacion.intef.es/pluginfile.php/43820/mod_imscp/content/8/representacin_de_un_algoritmo.html

[23] Hurtado, J. A. (s.f.). *Unicauca.* Obtenido de http://artemisa.unicauca.edu.co/~ahurtado/matematicas/Funcional.pdf

[25] Introducción a Scheme, http://www.dccia.ua.es/dccia/inf/asignaturas/LPP/2010-2011/teoria/tema2.html

[27] Corner, L. F. (8 de Febrero de 2012). *lluisfranco.com.* Obtenido de https://lluisfranco.com/2012/02/08/programacin-funcional-para-el-resto-de-nosotros/